Sound Regular Expression Semantics for Dynamic Symbolic Execution of JavaScript

Blake Loring
Information Security Group
Royal Holloway, University of London
United Kingdom
blake.loring.2015@rhul.ac.uk

Duncan Mitchell
Department of Computer Science
Royal Holloway, University of London
United Kingdom
duncan.mitchell.2015@rhul.ac.uk

Johannes Kinder
Research Institute CODE
Bundeswehr University Munich
Germany
johannes.kinder@unibw.de

Abstract

Support for regular expressions in symbolic execution-based tools for test generation and bug finding is insufficient. Common aspects of mainstream regular expression engines, such as backreferences or greedy matching, are ignored or imprecisely approximated, leading to poor test coverage or missed bugs. In this paper, we present a model for the complete regular expression language of ECMAScript 2015 (ES6), which is sound for dynamic symbolic execution of the `test` and `exec` functions. We model regular expression operations using string constraints and classical regular expressions and use a refinement scheme to address the problem of matching precedence and greediness. We implemented our model in ExpoSE, a dynamic symbolic execution engine for JavaScript, and evaluated it on over 1,000 Node.js packages containing regular expressions, demonstrating that the strategy is effective and can significantly increase the number of successful regular expression queries and therefore boost coverage.

CCS Concepts • **Software and its engineering → Software verification and validation**; **Dynamic analysis**; • **Theory of computation** → *Regular languages.*

Keywords Dynamic symbolic execution, JavaScript, regular expressions, SMT

ACM Reference Format:
Blake Loring, Duncan Mitchell, and Johannes Kinder. 2019. Sound Regular Expression Semantics for Dynamic Symbolic Execution of JavaScript. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3314221.3314645

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

1 Introduction

Regular expressions are popular with developers for matching and substituting strings and are supported by many programming languages. For instance, in JavaScript, one can write `/goo+d/.test(s)` to test whether the string value of `s` contains `"go"`, followed by one or more occurrences of `"o"` and a final `"d"`. Similarly, `s.replace(/goo+d/,"better")` evaluates to a new string where the first such occurrence in `s` is replaced with the string `"better"`.

Several testing and verification tools include some degree of support for regular expressions because they are so common [24, 27, 29, 34, 37]. In particular, SMT (satisfiability modulo theory) solvers now often support theories for strings and classical regular expressions [1, 2, 6, 15, 25, 26, 34, 38–40], which allow expressing constraints such as $\mathtt{s} \in \mathcal{L}(\mathtt{goo+d})$ for the `test` example above. Although any general theory of strings is undecidable [7], many string constraints are efficiently solved by modern SMT solvers.

SMT solvers support regular expressions in the language-theoretical sense, but "regular expressions" in programming languages like Perl or JavaScript—often called *regex*, a term we also adopt in the remainder of this paper—are not limited to representing regular languages [3]. For instance, the expression `/<(\w+)>.*?<\/\1>/` parses any pair of matching XML tags, which is a context-sensitive language (because the tag is an arbitrary string that must appear twice). Problematic features that prevent a translation of regexes to the word problem in regular languages include capture groups (the parentheses around `\w+` in the example above), backreferences (the `\1` referring to the capture group), and greedy/non-greedy matching precedence of subexpressions (the `.*?` is non-greedy). In addition, any such expression could also be included in a lookahead `(?=)`, which effectively encodes intersection of context sensitive languages. In tools reasoning about string-manipulating programs, these features are usually ignored or imprecisely approximated. This is a problem, because they are widely used, as we demonstrate in §7.1.

In the context of dynamic symbolic execution (DSE) for test generation, this lack of support can lead to loss of coverage or missed bugs where constraints would have to include membership in non-regular languages. The difficulty arises from the typical mixing of constraints in path conditions—simply *generating* a matching word for a standalone regex is

easy (without lookaheads). To date, there has been only limited progress on this problem, mostly addressing immediate needs of implementations with approximate solutions, e.g., for capture groups [29] and backreferences [27, 30]. However, neither matching precedence nor lookaheads have been addressed before.

In this paper, we propose a novel scheme for supporting ECMAScript regex in dynamic symbolic execution and show that it is effective in practice. We rely on the specification of regexes and their associated methods in ECMAScript 2015 (ES6). However, our methods and findings should be easily transferable to most other existing implementations. In particular, we make the following contributions:

- We fully model ES6 regex in terms of classical regular languages and string constraints (§4) and cover several aspects missing from previous work [27, 29, 30]. We introduce the notion of a *capturing language* to make the problem of matching and capture group assignment self-contained.
- We introduce a counterexample-guided abstraction refinement (CEGAR) scheme to address the effect of greediness on capture groups (§5), which allows us to deploy our model in DSE without sacrificing soundness for under-approximation.
- We present the first systematic study of JavaScript regexes, examining feature usage across 415,487 packages from the NPM software repository. We show that non-regular features are widely used (§7.1).

In the remainder of the paper we review ES6 regexes (§2) and present an overview of our approach by example (§3). We then detail our regex model using a novel formulation (§4), and we propose a CEGAR scheme to address matching precedence (§5). We discuss an implementation of the model as part of the ExpoSE symbolic execution engine for JavaScript (§6) and evaluate its practical impact on DSE (§7). Finally, we review related work (§8) and conclude (§9).

2 ECMAScript Regex

We review the ES6 regex specification, focusing on differences to classical regular expressions. We begin with the regex API and its matching behavior (§2.1) and then explain capture groups (§2.2), backreferences (§2.3), and operator precedence (§2.4). ES6 regexes are comparable to those of other languages but lack Perl's recursion and lookbehind and do not require POSIX-like longest matches.

2.1 Methods, Anchors, Flags

ES6 regexes are `RegExp` objects, created from literals or the `RegExp` constructor. `RegExp` objects have two methods, `test` and `exec`, which expect a string argument; String objects offer the `match`, `split`, `search` and `replace` methods that expect a `RegExp` argument.

A regex accepts a string if any portion of the string matches the expression, i.e., it is implicitly surrounded by wildcards. The relative position in the string can be controlled with *anchors*, with `^` and `$` matching the start and end, respectively.

Flags in regexes can modify the behavior of matching operations. The *ignore case* flag `i` ignores character cases when matching. The *multiline* flag `m` redefines anchor characters to match either the start and end of input or newline characters. The *unicode* flag `u` changes how unicode literals are escaped within an expression. The *sticky* flag `y` forces matching to start at `RegExp.lastIndex`, which is updated with the index of the previous match. Therefore, `RegExp` objects become stateful as seen in the following example:

```
r = /goo+d/y;
r.test("goood"); // true; r.lastIndex = 6
r.test("goood"); // false; r.lastIndex = 0
```

The meaning of the *global* flag `g` varies. It extends the effects of `match` and `replace` to include all matches on the string and it is equivalent to the sticky flag for the `test` and `exec` methods of `RegExp`.

2.2 Capture Groups

Parentheses in regexes not only change operator precedence (e.g., `(ab)*` matches any number of repetitions of the string `"ab"` while `ab*` matches the character `"a"` followed by any number of repetitions of the character `"b"`) but also create *capture groups*. Capture groups are implicitly numbered from left to right by order of the opening parenthesis. For example, `/a|((b)*c)*d/` is numbered as `/a|(`^{1}`(`^{2}`b)*c)*d/`. Where only bracketing is required, a non-capturing group can be created by using the syntax `(?: ...)`.

For regexes, capture groups are important because the regex engine will record the *most recent* substring matched against each capture group. Capture groups can be referred to from within the expression using backreferences (see §2.3). The last matched substring for each capture group is also returned by some of the API methods. In JavaScript, the return values of `match` and `exec` are arrays, with the whole match at index 0 (the implicit capture group 0), and the last matched instance of the i^{th} capture group at index i. In the example above, `"bbbbcbcd".match(/a|((b)*c)*d/)` will evaluate to the array `["bbbbcbcd", "bc", "b"]`.

2.3 Backreferences

A *backreference* in a regex refers to a numbered capture group and will match whatever the engine last matched the capture group against. In general, the addition of backreferences to regexes makes the accepted languages non-regular [3].

Inside quantifiers (Kleene star, Kleene plus, and other repetition operators), the string matched by the backreference can change across multiple matches. For example, the regex `/((a|b)\2)+/` can match the string `"aabb"`, with the backreference `\2` being matched twice: the first time, the capture

Table 1. Regular expression operators, separated by classes of precedence.

Operator	Name	Rewriting
(r)	Capturing parentheses	
$\backslash n$	Backreference	
$(?:r)$	Non-capturing parentheses	
$(?=r)$	Positive lookahead	
$(?!r)$	Negative lookahead	
\b	Word boundary	
\B	Non-word boundary	
$r*$	Kleene star	
$r*?$	Lazy Kleene star	
$r+$	Kleene plus	$r*r$
$r+?$	Lazy Kleene plus	$r*?r$
$r\{m,n\}$	Repetition	$r^n \mid \ldots \mid r^m$
$r\{m,n\}?$	Lazy repetition	$r^m \mid \ldots \mid r^n$
$r?$	Optional	$r \mid \epsilon$
$r??$	Lazy optional	$\epsilon \mid r$
$r_1 r_2$	Concatenation	
$r_1 \mid r_2$	Alternation	

group contains "a", the second time it contains "b". This logic applies recursively, and it is possible for backreferences to in turn be part of outer capture groups.

2.4 Operator Evaluation

We explain the operators of interest for this paper in Table 1; the implementation described in §6 supports the full ES6 syntax [14]. Some operators can be rewritten into semantically equivalent expressions to reduce the number of cases to handle (shown in the **Rewriting** column).

Regexes distinguish between *greedy* and *lazy* evaluation. Greedy operators consume as many characters as possible such that the entire regular expression still matches; lazy operators consume as few characters as possible. This distinction—called *matching precedence*—is unnecessary for classical regular languages, but does affect the assignment of capture groups and therefore backreferences.

Zero-length assertions or *lookarounds* do not consume any characters but still restrict the accepted word, enforcing a language intersection. Positive or negative *lookaheads* can contain any regex, including capture groups and backreferences. In ES6, *lookbehind* is only available through \b (word boundary), and \B (non-word boundary), which are commonly used to only (or never) match whole words in a string.

3 Overview

In an overview of our approach, we now define the word problem for regex (§3.1) and how it arises in DSE (§3.2). We introduce our model for regex by example (§3.3) and explain how to eliminate spurious solutions by refinement (§3.4).

```
1  let timeout = '500';
2  for (let i = 0; i < args.length; i++) {
3    let arg = args[i];
4    let parts = /<(\w+)>([0-9]*)<\/\1>/.exec(arg);
5    if (parts) {
6      if (parts[1] === "timeout") {
7        timeout = parts[2];
8      }
9      ...
10   }
11 }
12 assert(/^[0-9]+$/.test(timeout) == true);
```

Listing 1. Code example using regex

3.1 The Word Problem and Capturing Languages

For any given classical regular expression r, we write $w \in \mathcal{L}(r)$ whenever w is a word within the (regular) language generated by r. For a regex R, we also need to record values of capture groups within the regex. To this end, we introduce the following notion:

Definition 1 (Capturing Language)**.** The *capturing language* of a regex R, denoted $\mathcal{L}_c(R)$, is the set of tuples $(w, C_0, \ldots, C_n)$ such that w is a word of the language of R and each $C_0, \ldots, C_n$ is the substring of w matched by the corresponding numbered capture group in R.

A word w is therefore matched by a regex R if and only if $\exists C_0, \ldots, C_n : (w, C_0, \ldots, C_n) \in \mathcal{L}_c(R)$. It is not matched if and only if $\forall C_0, \ldots, C_n : (w, C_0, \ldots, C_n) \notin \mathcal{L}_c(R)$. For readability, we will usually omit quantifiers for capture variables where they are clear from the context.

3.2 Regex In Dynamic Symbolic Execution

The code in Listing 1 parses numeric arguments between XML tags from its input variable args, an array of strings. The regex in line 4 breaks each argument into two capture groups, the tag and the numeric value (parts[0] is the entire match). When the tag is "timeout", it sets the timeout value accordingly (lines 6–7). On line 12, a runtime assertion checks that the timeout value is truly numeric after the arguments have been processed. The assertion can fail because the program contains a bug: the regex in line 4 uses a Kleene star and therefore also admits the empty string as the number to set, and JavaScript's dynamic type system will allow setting timeout to the empty string.

DSE finds such bugs by systematically enumerating paths, including the failure branches of assertions [17]. Starting from a concrete run with input, say, args[0] = "foo", the DSE engine will attempt to build a *path condition* that encodes the branching decisions in terms of the input values. It then attempts to systematically flip clauses in the path condition and query an SMT solver to obtain input assignments

covering different paths. This process repeats forever or until all paths are covered (this program has an unbounded number of paths as it is looping over an input string).

Without support for regex, the DSE engine will *concretize* `arg` on the call to `exec`, assigning the concrete result to `parts`. With all subsequent decisions therefore concrete, the path condition becomes $pc = \texttt{true}$ and the engine will be unable to cover more paths and find the bug.

Implementing regex support ensures that `parts` is *symbolic*, i.e., its elements are represented as formulas during symbolic execution. The path condition for the initial path thus becomes $pc = (\texttt{args[0]}, C_0, C_1, C_2) \notin \mathcal{L}_c(R)$ where $R = \texttt{<(\w+)>([0-9]*)<\/\1>}$. Negating the only clause and solving yields, e.g., `args[0] = "<a>0</a>"`. DSE then uses this input assignment to cover a second path with $pc = (\texttt{args[0]}, C_0, C_1, C_2) \in \mathcal{L}_c(R) \wedge C_1 \neq \texttt{"timeout"}$. Negating the last clause yields, e.g., "`<timeout>0</timeout>`", entering line 7 and making `timeout` and therefore the assertion symbolic. This leads to $pc = (\texttt{args[0]}, C_0, C_1, C_2) \in \mathcal{L}_c(R) \wedge C_1 = \texttt{"timeout"} \wedge (C_2, C_0') \in \mathcal{L}_c(\texttt{^[0-9]+\$})$, which, after negating the last clause, triggers the bug with the input "`<timeout></timeout>`".

3.3 Modeling Capturing Language Membership

Capturing language membership constraints in the path condition cannot be directly expressed in SMT. We model these in terms of classical regular language membership and string constraints. For a given ES6 regex R, we first rewrite R (see Table 1) in atomic terms only, i.e., `|`, `*`, capture groups, backreferences, lookaheads, and anchors. For consistency with the JavaScript API, we also introduce the outer capture group C_0. Consider the regex $R = \texttt{(?:a|(b))\1}$. After preprocessing, the capturing language membership problem becomes

$$(w, C_0, C_1) \in \mathcal{L}_c(\texttt{(?:.|\n)*?((?:a|(b))\1)(?:.|\n)*?}),$$

a generic rewriting that allows for characters to precede and follow the match in the absence of anchors.

We recursively reduce capturing language membership to regular membership. To begin, we translate the purely regular Kleene stars and the outer capture group to obtain

$$\begin{aligned}(w, C_0, C_1) \in \mathcal{L}_c(R) \implies\ & w = w_1 \texttt{++}\ w_2 \texttt{++}\ w_3 \\ & \wedge\ w_1 \in \mathcal{L}(\texttt{(:?.|\n)*?}) \\ & \wedge\ (w_2, C_1) \in \mathcal{L}_c(\texttt{(?:a|(b))\1}) \wedge\ C_0 = w_2 \\ & \wedge\ w_3 \in \mathcal{L}(\texttt{(:?.|\n)*?}),\end{aligned}$$

where `++` is string concatenation. We continue by decomposing the regex until there are only purely regular terms or standard string constraints. Next, we translate the nested capturing language constraint

$$\begin{aligned}&(w_2, C_1) \in \mathcal{L}_c(\texttt{(?:a|(b))\1}) \implies \\ &\ w_2 = w_1' \texttt{++}\ w_2' \wedge (w_1', C_1) \in \mathcal{L}_c(\texttt{a|(b)}) \wedge (w_2') \in \mathcal{L}_c(\texttt{\1}).\end{aligned}$$

When treating the alternation, either the left is satisfied and the capture group becomes undefined (which we denote as $\varnothing$), or the right is satisfied and the capture is locked to the match, which we model as

$$(w_1' \in \mathcal{L}(a) \wedge C_1 = \varnothing) \vee (w_1' \in \mathcal{L}(b) \wedge C_1 = w_1').$$

Finally we model the backreference, which is case dependent on whether the capture group it refers to is defined or not:

$$(C_1 = \varnothing \implies w_2' = \epsilon) \wedge (C_1 \neq \varnothing \implies w_2' = C_1).$$

Putting this together, we obtain a model for R:

$$\begin{aligned}&(w, C_0, C_1) \in \mathcal{L}_c(R) \implies w = w_1 \texttt{++}\ w_1' \texttt{++}\ w_2' \texttt{++}\ w_3 \\ &\wedge\ C_0 = w_1' \texttt{++}\ w_2' \\ &\wedge\ \big((w_1' \in \mathcal{L}(a) \wedge C_1 = \varnothing) \vee (w_1' \in \mathcal{L}(b) \wedge C_1 = w_1')\big) \\ &\wedge\ (C_1 = \varnothing \implies w_2' = \epsilon) \wedge (C_1 \neq \varnothing \implies w_2' = C_1) \\ &\wedge\ w_1 \in \mathcal{L}(\texttt{(:?.|\n)*?}) \wedge w_3 \in \mathcal{L}(\texttt{(:?.|\n)*?}).\end{aligned}$$

3.4 Refinement

Because of matching precedence (greediness), these models permit assignments to capture groups that are impossible in real executions. For example, we model `/^a*(a)?$/` as

$$\begin{aligned}(w, C_0, C_1) \in \mathcal{L}_c(\texttt{/\^a*(a)?\$/}) \implies\ & w = w_1 \texttt{++}\ w_2 \\ & \wedge\ w_1 \in \mathcal{L}(\texttt{a*}) \wedge w_2 \in \mathcal{L}(\texttt{a}|\epsilon) \wedge C_0 = w \wedge C_1 = w_2.\end{aligned}$$

This allows C_1 to be either `a` or the empty string ϵ, i.e., the tuple `("aa", "aa", "a")` would be a spurious member of the capturing language under our model. Because `a*` is *greedy*, it will always consume both characters in the string `"aa"`; therefore, `(a)?` can only match ϵ. This problem posed by *greedy* and *lazy* operator semantics remains unaddressed by previous work [27, 29, 30, 34]. To address this, we use a counterexample-guided abstraction refinement scheme that validates candidate assignments with an ES6-compliant matcher. Continuing the example, the candidate element `("aa", "aa", "a")` is validated by running a concrete matcher on the string `"aa"`, which contradicts the candidate captures with $C_0 =$ `"aa"` and $C_1 = \epsilon$. The model is refined with the counter-example to the following:

$$\begin{aligned}w =\ & w_1 \texttt{++}\ w_2 \\ & \wedge\ w_1 \in \mathcal{L}(\texttt{a*}) \wedge w_2 \in \mathcal{L}(\texttt{a}|\epsilon) \wedge C_0 = w \wedge C_1 = w_2 \\ & \wedge\ \big(w = \texttt{"aa"} \implies (C_0 = \texttt{"aa"} \wedge C_1 = \epsilon)\big).\end{aligned}$$

We then generate and validate a new candidate (w, C_0, C_1) and repeat the refinement until a satisfying assignment passes the concrete matcher.

4 Modeling ES6 Regex

We now detail the process of modeling capturing languages. After preprocessing a given ES6 regex R to R' (§4.1), we model constraints $(w, C_0, \ldots, C_n) \in \mathcal{L}_c(R')$ by recursively translating terms in the abstract syntax tree (AST) of R'

to classical regular language membership and string constraints (§4.2–4.3). Finally, we show how to model negated constraints $(w, C_0, \ldots, C_n) \notin \mathcal{L}_c(R')$ (§4.4).

4.1 Preprocessing

For illustrative purposes, we make the concatenation R_1R_2 of terms R_1, R_2 explicit as the binary operator $R_1 \cdot R_2$. Any regex can then be split into combinations of atomic elements, capture groups and backreferences (referred to collectively as *terms*, in line with the ES6 specification [14]), joined by explicit operators. Using the rules in Table 1, we rewrite any R to an equivalent regex R' containing only alternation, concatenation, Kleene star, capture groups, non-capturing parentheses, lookarounds, and backreferences. We rewrite any remaining lazy quantifiers to their greedy equivalents, as the models are agnostic to matching precedence (this is dealt with in refinement).

Note that the rules for Kleene plus and repetition duplicate capture groups, e.g., rewriting `/(a){1,2}/` to `/(a)(a)|(a)/` adds two capture groups. We therefore explicitly relate capture groups between the original and rewritten expressions. When rewriting a Kleene plus expression $S+$ containing K capture groups, $S*S$ has $2K$ capture groups. For a constraint of the form $(C_1, \ldots, C_K) \in \mathcal{L}_c(S+)$, the rewriting yields

$$(C_0, C_{1,1}, \ldots, C_{K,1}, C_{1,2}, \ldots, C_{K,2}) \in \mathcal{L}_c(S*S).$$

Since $S*S$ contains two copies of S, $C_{i,j}$ corresponds to the i^{th} capture in the j^{th} copy of S in $S*S$. We express the direct correspondence between captures as

$$\begin{aligned}(w, C_0, C_1, \ldots, C_K) \in \mathcal{L}_c(S+) \iff \\ (w, C_0, C_{1,1}, \ldots, C_{K,1}, C_{1,2}, \ldots, C_{K,2}) \in \mathcal{L}_c(S*S) \\ \wedge \forall i \in \{1, \ldots, K\},\ C_i = C_{i,2}.\end{aligned}$$

For repetition, if $S\{m, n\}$ has K capture groups, then $S' = S^n \mid \ldots \mid S^m$ has $\frac{K}{2}(n+m)(n-m+1)$ captures. In S', suppose we index our captures as $C_{i,j,k}$ where $i \in \{1, \ldots, K\}$ is the index of the capture group in S, $j \in \{0, \ldots, n-m\}$ denotes which alternate the capture group is in, and $k \in \{0, \ldots, m + j-1\}$ indexes the copies of S within each alternate. Intuitively, we pick a single $x \in \{0, \ldots, n-m\}$ that corresponds to the first satisfied alternate. Comparing the assignment of captures in $S\{m, n\}$ to S', we know that the value of the capture is the last possible match, so $C_i = C_{i,x,m+x-1}$ for all $i \in \{1, \ldots, K\}$. Formally, this direct correspondence can be expressed as

$$\begin{aligned}&(w, C_0, C_1, \ldots, C_K) \in \mathcal{L}_c(S\{m, n\}) \iff \\ &\quad (w, C_0, C_{1,0,0}, \ldots, C_{K,n-m,n}) \in \mathcal{L}_c(S^n \mid \ldots \mid S^m) \\ &\quad \wedge \exists x \in \{0, \ldots, n-m\} : \\ &\quad \big((w, C_0, C_{1,x,0}, \ldots, C_{K,x,m+x-1}) \in \mathcal{L}_c(S^{m+x}) \\ &\qquad \wedge \forall x' > x,\ (w, C_0, C_{1,x',0}, \ldots, C_{K,x',m+x'-1}) \notin \mathcal{L}_c(S^{m+x'}) \\ &\qquad \wedge \forall i \in \{1, \ldots, K\},\ C_i = C_{i,x,m+x-1}\big).\end{aligned}$$

4.2 Operators and Capture Groups

Let t be the next term to process in the AST of R'. If t is capture-free and purely regular, there is nothing to do in this step. If t is non-regular, it contains $k+1$ capture groups (with $k \geq -1$) numbered i through $i+k$. At each recursive step, we express membership of the capturing language $(w, C_i, \ldots, C_{i+k}) \in \mathcal{L}_c(t)$ through a model consisting of string and regular language membership constraints, and a set of remaining capturing language membership constraints for subterms of t. Note that we record the locations of capture groups within the regex in the preprocessing step. When splitting t into subterms t_1 and t_2, capture groups $C_i, \ldots, C_{i+j}$ are contained in t_1 and $C_{i+j+1}, \ldots, C_{i+k}$ are contained in t_2 for some j. The models for individual operations are given in Table 2; we discuss specifics of the rules below.

When matching an alternation `|`, capture groups on the non-matching side will be undefined, denoted by $\varnothing$, which is distinct from the empty string ϵ.

When modeling quantification $t = t_1*$, we assume t_1 does not contain backreferences (we address this case in §4.3). In this instance, we model t via the expression $\hat{t_1}*t_1 \mid \epsilon$, where $\hat{t_1}$ is a regular expression corresponding to t_1, except each set of capturing parentheses is rewritten as a set of non-capturing parentheses. In this way, $\hat{t_1}$ is regular (it is backreference-free by assumption). However, $\hat{t_1}*t_1 \mid \epsilon$ is not semantically equivalent to t: if possible, capturing groups must be satisfied, so $\hat{t_1}*$ cannot consume all matches of the expression. We encode this constraint with the implication that $\hat{t_1}*$ must match the empty string whenever $t_1 \mid \epsilon$ does.

Lookahead constrains the word to be a member of the languages of both the assertion expression and t_2. The word boundary `\b` is effectively a single-character lookaround for word and non-word characters. Because the boundary can occur both ways, the model uses disjunction for the end of w_1 and the start of w_2 being word and non-word, or non-word and word characters, respectively. The non-word boundary `\B` is defined as the dual of `\b`.

For capture groups, we bind the next capture variable C_i to the string matched by t_1. The i^{th} capture group must be the outer capture and the remaining captures $C_{i+1}, \ldots, C_{i+k}$ must therefore be contained within t_1. There is nothing to be done for non-capturing groups and recursion continues on the contained subexpression.

Anchors assert the start (`^`) and end (`$`) of input; we represent the beginning and end of a word via the meta-characters $\langle$ and $\rangle$, respectively. In most instances when handling these operations, t_1 will be ϵ; this is because it is rare to have regex operators prior to those marking the start of input (or after marking the end of input, respectively). In both these cases, we assert that the language defines the start or end of input—and that as a result of this, the language of t_1 must be an empty word, though the capture groups may be defined (say through t_1 containing assertions with nested captures). We

Table 2. Models for regex operators.

Operation	t	**Overapproximate Model for** $(w, C_i, ..., C_{i+k}) \in \mathcal{L}_c(t)$
Alternation	$t_1 \mid t_2$	$\big((w, C_i, ..., C_{i+j}) \in \mathcal{L}_c(t_1) \wedge C_{i+j+1} = ... = C_{i+k} = \varnothing\big)$ $\vee \big((w, C_{i+j+1}, ..., C_{i+k}) \in \mathcal{L}_c(t_2) \wedge C_i = ... = C_{i+j} = \varnothing\big)$
Concatenation	$t_1 \cdot t_2$	$w = w_1 \mathbin{+\!\!+} w_2 \wedge (w_1, C_i, ..., C_{i+j}) \in \mathcal{L}_c(t_1) \wedge (w_2, C_{i+j+1}, ..., C_{i+k}) \in \mathcal{L}_c(t_2)$
Backreference-free Quantification	t_1*	$w = w_1 \mathbin{+\!\!+} w_2 \wedge w_1 \in \mathcal{L}(\hat{t_1}*) \wedge (w_2, C_i, ..., C_{i+k}) \in \mathcal{L}_c(t_1 \mid \epsilon)$ $\wedge \big(w_2 = \epsilon \implies (w_1 = \epsilon \wedge C_i = \ldots = C_{i+k} = \varnothing)\big)$
Positive Lookahead	$(?{=}t_1)\, t_2$	$(w, C_i, ..., C_{i+j}) \in \mathcal{L}_c(t_1 .*) \wedge (w, C_{i+j+1}, ..., C_{i+k}) \in \mathcal{L}_c(t_2)$
Negative Lookahead	$(?!{=}t_1)\, t_2$	$(w, C_i, ..., C_{i+j}) \notin \mathcal{L}_c(t_1 .*) \wedge (w, C_{i+j+1}, ..., C_{i+k}) \in \mathcal{L}_c(t_2)$
Input Start	$t_1\texttt{\^{}}$	$(w, C_i, ..., C_{i+k}) \in \mathcal{L}_c(t_1) \wedge (w, C_i, ..., C_{i+k}) \in \mathcal{L}(. * \langle)$
Input Start (Multiline)	$t_1\texttt{\^{}}$	$(w, C_i, ..., C_{i+k}) \in \mathcal{L}_c(t_1) \wedge (w, C_i, ..., C_{i+k}) \in \mathcal{L}(. * \langle \mid \backslash n)$
Input End	$\$t_1$	$(w, C_i, ..., C_{i+k}) \in \mathcal{L}_c(t_1) \wedge (w, C_i, ..., C_{i+k}) \in \mathcal{L}(\rangle .*)$
Input End (Multiline)	$\$t_1$	$(w, C_i, ..., C_{i+k}) \in \mathcal{L}_c(t_1) \wedge (w, C_i, ..., C_{i+k}) \in \mathcal{L}(\rangle \mid \backslash n .*)$
Word Boundary	$t_1 \backslash \texttt{b}\ t_2$	$w = w_1 \mathbin{+\!\!+} w_2 \wedge (w_1, C_i, ..., C_{i+j}) \in \mathcal{L}_c(t_1) \wedge (w_2, C_{i+j+1}, ..., C_{i+k}) \in \mathcal{L}_c(t_2)$ $\wedge \Big(\big((w_1 \in \mathcal{L}(.*\backslash \texttt{W}) \vee w_1 = \epsilon) \wedge w_2 \in \mathcal{L}(\backslash \texttt{w}.*)\big) \vee \big(w_1 \in \mathcal{L}(.*\backslash \texttt{w}) \wedge (w_2 \in \mathcal{L}(\backslash \texttt{W}.*) \vee w_2 = \epsilon)\big)\Big)$
Non-Word Boundary	$t_1 \backslash \texttt{B}\ t_2$	$w = w_1 \mathbin{+\!\!+} w_2 \wedge (w_1, C_i, ..., C_{i+j}) \in \mathcal{L}_c(t_1) \wedge (w_2, C_{i+j+1}, ..., C_{i+k}) \in \mathcal{L}_c(t_2)$ $\wedge \big((w_1 \notin \mathcal{L}(.*\backslash \texttt{W}) \wedge w_1 \neq \epsilon) \vee w_2 \notin \mathcal{L}(\backslash \texttt{w}.*)\big) \wedge \big(w_1 \notin \mathcal{L}(.*\backslash \texttt{w}) \vee (w_2 \notin \mathcal{L}(\backslash \texttt{W}.*) \wedge w_2 \neq \epsilon)\big)$
Capture Group	(t_1)	$(w, C_{i+1}, ..., C_{i+k}) \in \mathcal{L}_c(t_1) \wedge C_i = w$
Non-Capturing Group	$(?:t_1)$	$(w, C_i, ..., C_{i+k}) \in \mathcal{L}_c(t_1)$
Base Case	t regular	$w \in \mathcal{L}(t)$

give separate rules for matching a regular expression with the multiline flag set, which modify the behavior of anchors to accept either our meta-characters or a line break.

4.3 Backreferences

Table 3 lists our models for different cases of backreferences in the AST of regex R; $\backslash k$ is a backreference to the k^{th} capture group of R. Intuitively, each instance of a backreference is a variable that refers to a capture group and has a type that depends on the structure of R.

We call a backreference *immutable* if it can only evaluate to a single value when matching; it is *mutable* if it can take on multiple values, which is a rare but particularly tricky case. For example, consider `/((a|b)\2)+\1\2/`. Here, the backreference `\1` and the second instance of `\2` are immutable. However, the first instance of `\2` is mutable: each repetition of the outer capture group under the Kleene plus can change the value of the second (inner) capture group, in turn changing the value of the backreference inside this quantification. For example, the string `"aabbaabbb"` satisfies this regex, but `"aabaaabaa"` does not. To fully characterize these distinctions, we introduce the following definition:

Definition 2 (Backreference Type)**.** Let t be the k^{th} capture group of a regex R. Then

1. $\backslash k$ is *empty* if either k is greater than the number of capture groups in R, or $\backslash k$ is encountered before t in a post-order traversal of the AST of R;
2. $\backslash k$ is *mutable* if $\backslash k$ is not empty, and both t and $\backslash k$ are subterms of some quantified term Q in R;
3. otherwise, $\backslash k$ is *immutable*.

When a backreference is empty, it is defined as ϵ, because it refers to a capture group that either is a superterm, e.g., `/(a\1)*/`, or appears later in the term, e.g., `/\1(a)/`.

There are two cases for immutable backreferences. In the first case, the backreference is not quantified. In our model for R, C_k has already been modeled with an equality constraint, so we can bind the backreference to it. In the second case, the backreference occurs within a quantification; here, the matched word is a finite concatenation of identical copies of the referenced capture group. Both models also incorporate the corner case where the capture group is $\varnothing$ due to alternation or an empty Kleene star. Following the ES6 standard, the backreference evaluates to ϵ in this case.

Mutable backreferences appear in the form $(...t_1...\backslash k...)*$ where t_1 is the k^{th} capture group; ES6 does not support forward referencing of backreferences, so in $(...\backslash k...t_1...)*$, $\backslash k$ is empty. For illustration purposes, the fourth entry of Table 3 describes the simplest case for mutable backreferences, other patterns are straightforward generalizations. In this case, we

Table 3. Modeling backreferences.

Type of $\backslash k$	Capturing Language	Approximation	Model
Empty	$(w) \in \mathcal{L}_c(\backslash k)$	Exact	$w = \epsilon$
Immutable	$(w) \in \mathcal{L}_c(\backslash k)$	Overapproximate	$(C_k = \varnothing \implies w = \epsilon) \wedge (C_k \neq \varnothing \implies w = C_k)$
Immutable	$(w) \in \mathcal{L}_c(\backslash k*)$	Overapproximate	$(C_k = \varnothing \implies w = \epsilon) \wedge (C_k \neq \varnothing \implies \exists m \geq 0 : w = {+\!+}_{i=0}^{m} C_k)$
Mutable	$(w,C_k) \in \mathcal{L}_c((?:(t_1)\backslash k)*)$ t_1 is capture group-free	Overapproximate	$\big(w = \epsilon \wedge C_k = \varnothing\big) \vee \big(\exists m \geq 1 : w = {+\!+}_{i=1}^{m}(\sigma_{i,1} {+\!+} \sigma_{i,2})$ $\wedge \forall i > 1, \big((\sigma_{i,1}, C_{k,i}) \in \mathcal{L}_c(t_1) \wedge \sigma_{i,2} = C_{k,i}\big) \wedge C_k = C_{k,m}\big)$
Mutable	$(w,C_k) \in \mathcal{L}_c((?:(t_1)\backslash k)*)$ t_1 is capture group-free	Unsound	$\big(w = \epsilon \wedge C_k = \varnothing\big) \vee \big(\exists m \geq 1 : w = {+\!+}_{i=1}^{m}(\sigma_{i,1} {+\!+} \sigma_{i,2})$ $\wedge (\sigma_{i,1}, C_k) \in \mathcal{L}_c(t_1) \wedge \forall i \geq 1, (\sigma_{i,1} = \sigma_{1,1} \wedge \sigma_{i,2} = \sigma_{1,1})\big)$

assume t_1 is the k^{th} capture group but is otherwise capture group-free. We can treat the entirety of this term at once: as such, any word in the language is either ϵ, or for some number of iterations, we have the concatenation of a word in the language of t_1 followed by a copy of it. We introduce new variables $C_{k,i}$ referring to the values of the capture group in each iteration, which encodes the repeated matching on the string until settling on the final value for C_k. In this instance, we need not deal with the possibility that any $C_{k,i}$ is $\varnothing$, since the quantification ends as soon as t_1 does not match.

Unfortunately, constraints generated from this model are hard to solve and not feasible for current SMT solvers, because they require "guessing" a partition of the matched string variable into individual and varying components. To make solving such queries practical, we introduce an alternative to the previous rule where we treat quantified backreferences as immutable. The resulting model is shown in the last row of Table 3. E.g., returning to `/((a|b)\2)+\1\2/`, we accept (`"aaaaaaaaa"`, `"aaaaaaaaa"`, `"aaaa"`, `"a"`), but not (`"aabbaabbb"`, `"aabbaabbb"`, `"aabb"`, `"b"`). We discuss the soundness implications in §5.4. Quantified backreferences are rare (see §7.1), so the effect is limited in practice.

4.4 Modeling Non-Membership

The model described so far overapproximates membership of a capturing language. We define an analogous model for non-membership of the form $\forall C_0, \ldots, C_n : (w, C_0, \ldots, C_n) \notin \mathcal{L}_c(R)$. Intuitively, non-membership models assert that for all capture group assignments there exists some partition of the word such that one of the individual constraints is violated. Most models are simply negated. In concatenation and quantification, only language and emptiness constraints are negated, so the models take the form

$$\begin{aligned} w = w_1 {+\!+} w_2 \\ \wedge\big(\ldots \notin \mathcal{L}_c(\ldots) \vee \ldots \notin \mathcal{L}_c(\ldots) \\ \vee (w_2 = \epsilon \wedge \neg(w_1 = \epsilon \ldots))\big). \end{aligned}$$

In the same manner, the model for capture groups is

$$(w, C_{i+1}, ..., C_{i+k}) \notin \mathcal{L}_c(t_1) \wedge C_i = w.$$

Returning to the example of §3.3, the negated model for $\forall C_0, C_1 : (w, C_0, C_1) \notin \mathcal{L}_c((?:a|(b))\backslash 1)$ becomes

$$\begin{aligned} &\forall C_0, C_1 : w = w_1 {+\!+} w_1' {+\!+} w_2' {+\!+} w_3 \\ &\wedge C_0 = w_1' {+\!+} w_2' \\ &\wedge \big(\neg\big((w_1' \in \mathcal{L}(a) \wedge C_1 = \varnothing) \vee (w_1' \in \mathcal{L}(b) \wedge C_1 = w_1')\big) \\ &\quad \vee \neg(C_1 = \varnothing \implies w_2' = \epsilon) \vee \neg(C_1 \neq \varnothing \implies w_2' = C_1) \\ &\quad \vee w_1 \notin \mathcal{L}((:?.|\backslash n)*?) \vee w_3 \notin \mathcal{L}((:?.|\backslash n)*?)\big). \end{aligned}$$

5 Matching Precedence Refinement

We now explain the issue of matching precedence (§5.1) and introduce a counterexample-guided abstraction refinement scheme (§5.2) to address it. We discuss termination (§5.3) and the overall soundness of our approach (§5.4).

5.1 Matching Precedence

The model in Tables 2 and 3 does not account for matching precedence (see §3.4). A standards-compliant ES6 regex matcher will derive a unique set of capture group assignments when matching a string w, because matching precedence dictates that greedy (non-greedy) expressions match as many (as few) characters as possible before moving on to the next [14]. These requirements are not part of our model, as encoding them directly into SMT would require nesting of quantifiers for each operator, making them impractical for automated solving.

5.2 CEGAR for ES6 Regular Expression Models

We eliminate infeasible elements of the capturing language admitted by our model through counter example-guided abstraction refinement (CEGAR).

Algorithm 1 is a CEGAR-based satisfiability checker for constraints modeled from ES6 regexes, which relies on an external SMT solver with classical regular expression and string support and an ES6-compliant regex matcher. The algorithm takes an SMT problem P (derived from the DSE path condition) as a conjunction of constraints, some of

Algorithm 1: Counterexample-guided abstraction refinement scheme for matching precedence.

Input : Constraint problem P including models for m constraints $(w_j, C_{0,j}, \ldots, C_{n_j,j}) \boxdot_j \mathcal{L}_c(R_j)$.
Output : `null` if P is unsatisfiable, or a satisfying assignment for P otherwise

```
1  M := null;
2  Failed := false;
3  do
4      M := Solve(P);
5      if M = null then
6          return null;
7      Failed := false;
8      for j := 0 to m − 1 do
9          (C^♮_{0,j}, ..., C^♮_{n_j,j}) := ConcreteMatch(M[w_j], R_j) ;
10         if (C^♮_{0,j}, ..., C^♮_{n_j,j}) then
11             if ⊡_j = ∈ then
12                 for i := 0 to n_j do
13                     if C^♮_{i,j} ≠ M[C_{i,j}] then
14                         Failed := true;
15                         P := P ∧ (w_j = M[w_j] ⟹
                               ⋀_{0≤i≤n_j} C_{i,j} = C^♮_{i,j}) ;
16                 else                    // Non-membership query
17                     Failed := true;
18                     P := P ∧ (w_j ≠ M[w_j]);
19             else                        // No concrete match
20                 if ⊡_j = ∈ then
21                     Failed := true;
22                     P := P ∧ (w_j ≠ M[w_j]);
23 while Failed;
24 return M;
```

which model the $m \geq 0$ original capturing language membership constraints. We number the original capturing language constraints $0 \leq j < m$ so that we can refer to them as $(w_j, C_{0,j}, \ldots, C_{n_j,j}) \boxdot_j \mathcal{L}_c(R_j)$, where $\boxdot \in \{\in, \notin\}$. The algorithm returns `null` if P is unsatisfiable, or a satisfying assignment with correct matching precedence.

In a loop, we first pass the problem P to an external SMT solver. The solver returns a satisfying assignment M or `null` if the problem is unsatisfiable, in which case we are done (lines 4–6). If M is not null, the algorithm uses a concrete regular expression matcher (e.g., Node.js's built-in matcher) to populate concrete capture variables $C^{\natural}_{i,j}$ corresponding to the words w_j in M.

Lines 8–22 describe how the assignments of capture groups are checked for each regular expression R_j in the original problem P. We first check whether the concrete matcher returned a list of valid capture group assignments, i.e., whether the word $M[w_j]$ from the satisfying assignment matches concretely. If it did, then w_j is a member of the language generated by R_j. If $\boxdot_j = \in$, i.e., the membership constraint was positive, then we must check if the capture group assignments are consistent with those from M (line 13). If they are, we move on to the next regex, otherwise we refine the constraint problem by fixing capture group assignments to their concrete values for the matched word (line 15). Dually, if a modeled non-membership constraint was satisfiable but the word from the current satisfying assignment $M[w_j]$ did match concretely, we refine the problem by asserting that w must not equal that word (line 18). We do the same if $M[w_j]$ did not match concretely but came from a satisfied positive membership constraint (line 22).

If no refinement was necessary we have confirmed the overall assignment satisfies P and return M (line 24). Otherwise, the loop continues with solving the refined problem.

5.3 Termination

Unsurprisingly, CEGAR may require arbitrarily many refinements on pathological formulas and never terminate. This is unavoidable due to undecidability [7]. In practice, we therefore impose a limit on the number of refinements, leading to *unknown* as a possible third result. SMT solvers already may timeout or report *unknown* for complex string formulas, so this does not lead to additional problems in practice.

5.4 Soundness

When constructing the rules in Tables 2 and 3, we followed the semantics of regular expressions as laid out in the ES6 standards document [14]. The ES6 standard is written in a semi-formal fashion, so we are confident that our translation into logic is accurate, but cannot have formal proof. Existing attempts to encode ECMAScript semantics into logic such as JSIL [8] or KJS [28] do not include regexes.

With the exception of the optimized rule for mutable backreferences, our models are overapproximate, because they ignore matching precedence. When the CEGAR loop terminates, any spurious solutions from overapproximation are eliminated. As a result, we have an *exact* procedure to decide (non)-membership for capturing languages of ES6 regexes without quantified backreferences.

In the presence of quantified backreferences, the model after CEGAR termination becomes *underapproximate*. Since DSE itself is an underapproximate program analysis (due to concretization, solver timeouts, and partial exploration), our model and refinement strategy are *sound for DSE*.

6 Implementation

We now describe an implementation of our approach in the DSE engine ExpoSE[1] [27]. We explain how to model the regex API with capturing language membership (§6.1) and give a brief overview of ExpoSE (§6.2).

[1]ExpoSE is available at https://github.com/ExpoSEJS/ExpoSE.

Algorithm 2: RegExp.exec(input)

```
1  input′ := '⟨' + input + '⟩';
2  if sticky or global then
3      offset := lastIndex > 0 ? lastIndex + 1 : 0;
4      input′ := input′.substring(offset);
5  source′ := '(:?.|\n)*?(' + source + ')(:?.|\n)*?';
6  if caseIgnore then
7      source′ := rewriteForIgnoreCase(source′);
8  if (input′, C0, ..., Cn) ∈ Lc(source′) then
9      Remove ⟨ and ⟩ from (input′, C0, ..., Cn);
10     lastIndex := lastIndex + C0.startIndex + C0.length;
11     result := [C0, ..., Cn];
12     result.input := input;
13     result.index := C0.startIndex;
14     return result;
15 else
16     lastIndex := 0;
17     return undefined;
```

6.1 Modeling the Regex API

The ES6 standard specifies several methods that evaluate regexes [14]. We follow its specified pseudocode for `RegExp.exec(s)` to implement matching and capture group assignment in terms of capturing language membership in Algorithm 2. Notably, our algorithm implements support for all flags and operators specified for ES6.

`RegExp.test(s)` is precisely equivalent to the expression `RegExp.exec(s) !== undefined`. In the same manner, one can construct models for other regex functions defined for ES6. Our implementation includes partial models for the remaining functions that allow effective test generation in practice but are not semantically complete.

Algorithm 2 first processes flags to begin from the end of the previous match for sticky or global flags, and it rewrites the regex to accept lower and upper case variants of characters for the ignore case flag.

We introduce the ⟨ and ⟩ meta-characters to *input* which act as markers for the start and end of a string during matching. Next, if the sticky or global flags are set we slice *input* at `lastIndex` so that the new match begins from the end of the previous. Due to the introduction of our meta-characters thel `lastIndex` needs to be offset by 1 if it is greater than zero. We then rewrite the regex source to allow for characters to precede and succeed the match. Note that we use `(?:.|n)*?` rather than `.*?` because the wildcard `.` consumes all characters except line breaks in ECMAScript regexes. To avoid adding these characters to the final match we place the original regex source inside a capture group. This forms C_0, which is defined to be the whole matched string [14]. Once preprocessing is complete we test whether the input string and fresh string for each capture group are within the capturing language for the expression. If they are then a results object is created which returns the correctly mapped capture groups, the input string, and the start of the match in the string with the meta-characters removed. Otherwise `lastIndex` is reset and `undefined` is returned.

6.2 ExpoSE

ExpoSE is a DSE engine which uses the Jalangi2 [19] framework to instrument a piece of JavaScript software in order to create a program trace. As the program terminates, ExpoSE calls the SMT solver Z3 [13] to identify all feasible alternate test-cases from the trace. These new test cases are then queued and the next test case is selected for execution, in the manner of generational search [18]. The ExpoSE framework allows for the parallel execution of individual test cases, aggregating coverage and alternative path information as each test case terminates. This parallelization is achieved by executing each test case as a unique process allocated to a dedicated single core; as such the analysis is highly scalable.

Our strategy for test case selection is similar to the CUPA strategy proposed by Bucur et al. [9]. We use program fork points to prioritize unexplored code: each expression is given a unique identifier and scheduled test cases are sorted into buckets based upon which expression was being executed when they were created. We select the next test case by choosing a random test case from the bucket that has been accessed least during the analysis; this prioritizes test cases triggered by less common expressions.

7 Evaluation

We now empirically answer the following research questions:

- **(RQ1)** Are non-classical regexes an important problem in JavaScript?
- **(RQ2)** Does accurate modeling of ES6 regexes make DSE-based test generation more effective?
- **(RQ3)** Does the performance of the model and the refinement strategy enable practical analysis?

We answer the first question with a survey of regex usage in the wild (§7.1). We address RQ2 by comparing our approach against an existing partial implementation of regex support in ExpoSE [27] on a set of widely used libraries (§7.2). We then measure the contribution of each aspect of our approach on over 1,000 JavaScript packages (§7.3). We answer RQ3 by analyzing solver and refinement statistics per query (§7.4).

7.1 Surveying Regex Usage

We focus on code written for Node.js, a popular framework for standalone JavaScript. Node.js is used for both server and desktop applications, including popular tools *Slack* and *Skype*. We analyzed 415,487 packages from the NPM repository, the primary software repository for open source Node.js code. Nearly 35% of NPM packages contain a regex, 20% contain a capture group and 4% contain a backreference.

Table 4. Regex usage by NPM package.

Feature	Count	%
Packages on NPM	415,487	100.0%
. . . with source files	381,730	91.9%
. . . with regular expressions	145,100	34.9%
. . . with capture groups	84,972	20.5%
. . . with backreferences	15,968	3.8%
. . . with quantified backreferences	503	0.1%

Table 5. Feature usage by unique regex.

Feature	Total	%	Unique	%
Total Regex	9,552,546	100%	305,691	100%
Capture Groups	2,360,178	24.71%	119,051	38.94%
Global Flag	2,620,755	27.44%	90,356	29.56%
Character Class	2,671,565	27.97%	71,040	23.24%
Kleene+	1,541,336	16.14%	67,508	22.08%
Kleene*	1,713,713	17.94%	66,526	21.76%
Ignore Case Flag	1,364,526	14.28%	58,831	19.25%
Ranges	1,273,726	13.33%	52,155	17.06%
Non-capturing	1,236,533	12.94%	25,946	8.49%
Repetition	360,578	3.7%	17,068	5.58%
Kleene* (Lazy)	230,060	2.41%	13,250	4.33%
Multiline Flag	137,366	1.44%	10,604	3.47%
Word Boundary	336,821	3.53%	9,677	3.17%
Kleene+ (Lazy)	148,604	1.56%	6,072	1.99%
Lookaheads	176,786	1.85%	3,123	1.02%
Backreferences	64,408	0.67%	2,437	0.80%
Repetition (Lazy)	2,412	0.03%	221	0.07%
Quantified BRefs	1,346	0.01%	109	0.04%
Sticky Flag	98	<0.01%	60	0.02%
Unicode Flag	73	<0.01%	48	0.02%

Methodology We developed a lightweight static analysis that parses all source files in a package and identifies regex literals and function calls. We do not detect expressions of the form `new RegExp(...)`, as they would generally require a more expensive static analysis. Our numbers therefore provide a lower bound for regex usage.

Results We found regex usage in JavaScript to be widespread, with 145,100 packages containing at least one regex out of a total 415,487 scanned packages. Table 4 lists the number of NPM packages containing regexes, capture groups, backreferences, and backreferences appearing within quantification. Note that a significant number of packages make use of capture groups and backreferences, confirming the importance of supporting them.

Table 5 reports statistics for all 9M regexes collected, giving for each feature the fraction of expressions including it. Many regexes in NPM packages are not unique; this appears to be due to repeated inclusion of the same literal (instead of introduction of a constant), the use of online solutions to common problems, and the inclusion of dependencies (foregoing proper dependency management). To adjust for this, we provide data for both all expressions encountered and for just unique expressions. In both cases, there are significant numbers of capture groups, backreferences, and other non-classical features. As the occurrence rate of quantified backreferences is low, we do not differentiate between mutable and immutable backreferences.

Conclusions Our findings confirm that regexes are widely used and often contain complex features. Of particular importance is a faithful treatment of capture groups, which appear in 20.45% of the packages examined. On the flip side, since quantified backreferences make up just 0.01% of regexes, the optimization introduced in §4.3 will rarely lead to additional underapproximation during DSE.

7.2 Improvement Over State of the Art

We compare our approach against the original ExpoSE [27], which is, to our knowledge, the only available and functional implementation of regex support in JavaScript.

Methodology We evaluated statement coverage achieved by both versions of ExpoSE on a set of libraries, which we chose for their popularity (with up to 20M weekly downloads) and use of regex. This includes the three libraries minimist, semver, and validator, which the first version of ExpoSE was evaluated on [27]. To fairly compare original ExpoSE against our extension, we use the original automated library harness for both. Therefore we do not take advantage of other improvements for test generation, such as symbolic array support, which we have added in the course of our work. We re-executed each package six times for one hour each on both versions, using 32-core machines with 256GB of RAM, and averaged the results. We limited the refinement scheme to 20 iterations, which we identified as effective in preliminary testing (see §7.4).

Results Table 6 contains the results of our comparison. To provide an indication of program size, we use the number of lines of code loaded at runtime (JavaScript's dynamic method of loading dependencies makes it hard to determine a meaningful LOC count statically).

The results demonstrate that ExpoSE extended with our model and refinement strategy can improve coverage more than tenfold on our sample of widely-used libraries. In the cases of moment, query-string, and yn, the lack of ES6 support in the original ExpoSE prohibited meaningful analysis, leading to 0% coverage. In the case of semver, we see a decrease in coverage if stopped after one hour. This is due to the modeling of regex increasing solving time (see also §7.4). The coverage deficit disappears when executing both versions of ExpoSE with a timeout of two hours.

Table 6. Statement coverage with our approach (**New**) vs. [27] (**Old**) and the relative increase (+) on popular NPM packages (**Weekly** downloads). **LOC** are lines loaded and **RegEx** are regular expression functions symbolically executed.

Library	Weekly	LOC	RegEx	Old(%)	New(%)	+(%)
babel-eslint	2,500k	23,047	902	21.0	26.8	27.6
fast-xml-parser	20k	706	562	3.1	44.6	1,338.7
js-yaml	8,000k	6,768	78	4.4	23.7	438.6
minimist	20,000k	229	72,530	65.9	66.4	0.8
moment	4,500k	2,572	21	0.0	52.6	∞
query-string	3,000k	303	50	0.0	42.6	∞
semver	1,800k	757	616	51.7	46.2	−10.6
url-parse	1,400k	322	448	60.9	71.8	17.9
validator	1,400k	2,155	94	67.5	72.2	7.0
xml	500k	276	1,022	60.2	77.5	28.7
yn	700k	157	260	0.0	54.0	∞

Conclusions We find that our modifications to ExpoSE make test generation more effective in widely used libraries using regex. This suggests that the new method of solving regex queries presented in this paper has a substantial impact on practical problems in DSE. We also see that other improvements to ExpoSE, such as ES6 support, have affected coverage. Therefore, we continue with an evaluation of the individual aspects of our model.

7.3 Breakdown of Contributions

We now drill down into how the individual improvements in regex support are contributing to increases in coverage.

Methodology From the packages with regexes from our survey §7.1, we developed a test suite of 1,131 NPM libraries for which ExpoSE is able to automatically generate a meaningful test harness. In each of the libraries selected, ExpoSE executed at least one regex operation on a symbolic string, which ensures that the library contains some behavior relevant to the scope of this paper. The test suite constructed in this manner contains numerous libraries that are dependencies of packages widely used in industry, including Express and Lodash.[2]

Automatic test generation typically requires a bespoke test harness or set of parameterized unit tests [33] to achieve high coverage in code that does not have a simple command line interface, including libraries. ExpoSE's harness explores libraries fully automatically by executing all exported methods with symbolic arguments for the supported types `string`, `boolean`, `number`, `null` and `undefined`. Returned objects or functions are also subsequently explored in the same manner.

We executed each package for one hour, which typically allowed to reach a (potentially initial) coverage plateau, at which additional test cases do not increase coverage further. We break down our regex support into four levels and measure the contribution and cost of each one to line coverage and test execution rate (Table 7). As baseline, we first execute all regex methods concretely, concretizing the arguments and results. In the second configuration, we add the model for ES6 regex and their methods, including support for word boundaries and lookaheads, but remove capture groups and concretize any accesses to them, including backreferences. Third, we also enable full support for capture groups and backreferences. Fourth, we finally also add the refinement scheme to address overapproximation.

[2]Raw data for the experiments, including all package names, is available at https://github.com/ExpoSEJS/PLDI19-Raw-Data.

Table 7. Breakdown of how different components contribute to testing 1,131 NPM packages, showing number (#) and fraction (%) of packages with coverage improvements, the geometric mean of the relative coverage increase from the feature (**Cov**), and test execution rate.

Regex Support Level	Improved #	Improved %	Cov +(%)	Tests/min
Concrete Regular Expressions	-	-	-	11.46
+ Modeling RegEx	528	46.68%	+6.16%	10.14
+ Captures & Backreferences	194	17.15%	+4.18%	9.42
+ Refinement	63	5.57%	+4.17%	8.70
All Features vs. Concrete	617	54.55%	+6.74%	

Results Table 7 shows, for each level of support, the number and percentage of target packages where coverage improved; the geometric mean of the relative increase in coverage; and the mean test execution rate. The final row shows the effect of enabling full support compared to the baseline. Note that the number of packages improved is less than the sum of the rows above, since the coverage of a package can be improved by multiple features.

In a dataset of this size that includes many libraries that make only little use of regex, average coverage increases are expected to be small. Nevertheless, we see that dedicated support improves the coverage of more than half of packages that symbolically executed at least one regex function. As expected, the biggest improvement comes from supporting basic symbolic execution of regular expressions, even without capture groups or regard for matching precedence. However, we see further improvements when adding capture groups, which shows that they indeed affect program semantics. Refinement affects fewer packages, although it significantly contributes to coverage where it is required. This is because a lucky solver may generate correct inputs on the first attempt, even in ambiguous settings.

On some libraries in the dataset, the approach is highly effective. For example, in the manifest parser *n4mf-parser*, full support improves coverage by 29% over concrete; in the format conversion library *sbxml2json*, by 14%; and in the

Table 8. Solver times per package and query.

	Constraint Solver Time		
Packages/Queries	**Minimum**	**Maximum**	**Mean**
All packages	0.04s	12h 15m	2h 34m
With capture groups	0.20s	12h 15m	2h 40m
With refinement	0.46s	12h 15m	2h 48m
Where refinement limit is hit	3.49s	11h 07m	3h 17m
All queries	0.001s	22m 26s	0.15s
With capture groups	0.001s	22m 26s	5.53s
With refinement	0.005s	18m 51s	22.69s
Where refinement limit is hit	0.120s	18m 51s	58.85s

browser detection library *mario*, by 16%. In each of these packages the refinement scheme contributed to the improvement in coverage. In general, the largest increases are seen in packages that include regular expression-based parsers.

Each additional feature causes a small decrease in average test execution rate. Although a small fraction (~1%) of queries can take longer than 300s to solve, concurrent test execution prevents DSE from stalling on a single query.

Conclusions Full support for ES6 regex improves performance of DSE of JavaScript in practice at a cost of a 16% increase in execution time (RQ2). An increase in coverage at lower execution rate in a fixed time window suggests that full regular expression support increases the quality of individual test cases.

7.4 Effectiveness on Real-World Queries

We now investigate the performance of the model and refinement scheme to answer RQ3. Finally, we also discuss the refinement limit and how it affects analysis.

Methodology We collected data on queries during the NPM experiments (§7.3) to provide details on SMT query success rates and execution times, as well as on the usage of the refinement scheme.

Results We found that 753 (66%) of the 1,131 packages tested executed at least one query containing a capture group or backreference. Of these packages, 653 (58% overall) contained at least one query to the SMT solver requiring refinement, and 134 (12%) contained a query that reached the refinement limit.

In total, our experiments executed 58,390,184 SMT queries to generate test cases. As expected, the majority do not involve regexes, but they form a significant part: 4,489,581 (7.6%) queries modeled a regex, 645,295 (1.1%) modeled a capture group or backreference, 74,076 (0.1%) required use of the refinement scheme and 2,079 (0.003%) hit the refinement limit. The refinement scheme was overwhelmingly effective: only 2.8% of queries with at least one refinement also reached the refinement limit (0.003% of all queries where a capture group was modeled). Of the refined SMT queries, the mean number of refinements required to produce a valid satisfying assignment was 2.9; the majority of queries required only a single refinement.

Table 8 details time spent processing SMT problems per-package and per-query. We provide the data over the four key aspects of the problem: we report the time spent in the constraint solver both per package and per query in total, as well as the time in the constraint solver for the particularly challenging parts of our strategy. We found that the use of refinements increased the average per-query solving time by a factor of four; however, this is dominated by SMT queries that hit the refinement limit, which took ten times longer to run on average. The low minimum time spent in the solver in some packages can be attributed to packages where a regular expression was encountered early in execution but limitations in the test harness or function models (unrelated to regular expressions) prevented further exploration.

Conclusions We find the refinement scheme is highly effective, as it is able to solve 97.2% of encountered constraint problems containing regexes. It is also necessary, as 10% of queries containing a capture group had led to a spurious satisfying assignment and required refinement.

Usually, only a small number of refinements are required to produce a correct satisfying assignment. Therefore, even refinement limits of five or fewer are feasible and may improve performance with low impact on coverage.

7.5 Threats to Validity

We now look at potential issues affecting the validity of our results, in particular soundness, package selection, and scalability.

Soundness In addition to soundness of the model (see §5.4), one must consider the soundness of the implementation. In the absence of a mechanized specification for ES6 regex, our code cannot be proven correct, so we use an extensive test suite for validation. However, assuming the concrete matcher is specification-compliant, Algorithm 1 will, if it terminates, return a specification-compliant model of the constraint formula even if the implementation of §4 contains bugs. In the worst case, the algorithm would not terminate, leading to timeouts and loss of coverage. Bugs could therefore only have lowered the reported coverage improvements.

Package Selection and Harness In §7.3, we chose packages identified in our survey (§7.1) where our generic harness encountered a regular expression within one hour of DSE. This allowed us to focus the evaluation on regex support as opposed to evaluating the quality of the harness (and having to deal with unreachable code in packages). Use of this harness may have limited package selection to simpler, unrepresentative libraries. However, we found that simple APIs do not imply simple code: the final dataset contains several

complex packages, such as language parsers, and the types of regexes encountered were in line with the survey results. On simple code we found that ExpoSE would often reach 100% coverage; failure to do so was either due to the complexity of the code or the lack of support for language features unrelated to regex and APIs that would require additional modeling (e.g., the file system).

Scalability Scalability is a challenge for DSE in general, and is not specific to our model for regex. Empirically, execution time for a single test (instrumentation, execution, and constraint generation) grows linearly with program size, as does the average size of solver queries. The impact of query length on solving time varies, but does not appear to be exacerbated by our regex model. In principle, our model is compatible with compositional approaches [4, 16] and state merging [5, 21], which can help DSE scale to large programs.

The scalability of our approach suffices for Node.js, however: JavaScript has smaller LOC counts than, e.g., C++, and code on NPM is very modular. For instance, among the top 25 most depended-upon NPM libraries, the largest is 30 KLOC (but contains no regex). Several packages selected for our evaluation, such as babel-eslint, had between 20-30 KLOC and were meaningfully explored with the generic harness.

8 Related Work

In prior work, we introduced ExpoSE and partial support for encoding JavaScript regex in terms of classical regular language membership and string constraints [27]. This initial take on the problem was lacking support for several problematic features such as lookaheads, word boundaries, and anchors. Matching precedence was presented as an open problem, which we have now addressed through our refinement scheme.

In theory, regex engines can be symbolically executed themselves through the interpreter [9]. While this removes the need for modeling, in practice the symbolic execution of the entire interpreter and regex engine quickly becomes infeasible due to path explosion.

There have been several other approaches for symbolic execution of JavaScript; most include some limited support for classical regular expressions. Li et al. [24] presented an automated test generation scheme for programs with regular expressions by on-line generation of a matching function for each regular expression encountered, exacerbating path explosion. Saxena et al. [29] proposed the first scheme to encode capture groups through string constraints. Sen et al. [31] presented Jalangi, a tool based on program instrumentation and concolic values. Li and Ghosh [23] and Li et al. [22] describe a custom browser and symbolic execution engine for JavaScript and the browser DOM, and a string constraint solver *PASS* with support for most JavaScript string operations. Although all of these approaches feature some support for ECMAScript regex (such as limited support for capture groups), they ignore matching precedence and do not support backreferences or lookaheads.

Thomé et al. [32] propose a heuristic approach for solving constraints involving unsupported string operations. We choose to model operations unsupported by the solver and employ a CEGAR scheme to ensure correctness. Abdulla et al. [2] propose the use of a refinement scheme to solve complex constraint problems, including support for context-free languages. The language of regular expressions with backreferences is not context-free [10] and, as such, their scheme does not suffice for encoding all regexes; however, their approach could serve as richer base theory than classic regular expressions. Scott et al. [30] suggest backreferences can be eliminated via concatenation constraints, however they do not present a method for doing so.

Further innovations from the string solving community, such as work on the decidability of string constraints involving complex functions [12, 20] or support for recursive string operations [35, 36], are likely to improve the performance of our approach in future. We incorporate our techniques at the level of the DSE engine rather than the constraint solver, which allows our tool to leverage advances in string solving techniques; at the same time, we can take advantage of the native regular expression matcher and can avoid having to integrate implementation language-specific details for regular expressions into the solver.

A previous survey of regex usage across 4,000 Python applications [11] also provides a strong motivation for modeling regex. Our survey extends this work to JavaScript on a significantly larger sample size.

9 Conclusion

In this paper we presented a model for the complete regex language of ES6, which is sound for the dynamic symbolic execution of the `test` and `exec` functions. We model regex membership constraints in terms of string constraints and classical regular language membership. We introduced a novel CEGAR scheme to address the challenge of matching precedence, which so far had been largely ignored in related work. To the best of our knowledge, ours is the first comprehensive solution for ES6. We demonstrated that regexes—and specifically their non-regular features—are extensively used in JavaScript and that existing DSE-based analyses would therefore suffer coverage loss from concretization. In a large scale evaluation of over 1,000 Node.js programs, our novel solution outperforms existing partial approaches to the problem and demonstrates the viability of our model for improving the analysis of string-manipulating JavaScript programs.

Acknowledgments

Blake Loring was supported by the EPSRC Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/K035584/1).

References

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2015. Norn: An SMT Solver for String Constraints. In *Computer Aided Verification (CAV)*.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukáš Holík, Ahmed Rezine, and Philipp Rümmer. 2017. Flatten and Conquer: A Framework for Efficient Analysis of String Constraints. In *ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*.

[3] Alfred V. Aho. 1990. Algorithms for Finding Patterns in Strings. In *Handbook of Theoretical Computer Science (Vol. A)*, Jan van Leeuwen (Ed.). MIT Press, 255–300.

[4] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-driven Compositional Symbolic Execution. In *14th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[5] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing symbolic execution with Veritesting. In *36th Int. Conf. Software Engineering (ICSE)*. 1083–1094.

[6] Nikolaj Bjørner, Vijay Ganesh, Raphaël Michel, and Margus Veanes. 2012. SMT-LIB Sequences and Regular Expressions. In *Int. Workshop on Satisfiability Modulo Theories (SMT)*.

[7] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. 2009. Path Feasibility Analysis for String-Manipulating Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[8] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL)*.

[9] Stefan Bucur, Johannes Kinder, and George Candea. 2014. Prototyping symbolic execution engines for interpreted languages. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[10] Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. 2003. A Formal Study of Practical Regular Expressions. *Int. J. Foundations of Computer Science* 14, 06 (2003).

[11] Carl Chapman and Kathryn T. Stolee. 2016. Exploring Regular Expression Usage and Context in Python. In *Int. Symp. on Software Testing and Analysis (ISSTA)*.

[12] Taolue Chen, Yan Chen, Matthew Hague, Anthony W. Lin, and Zhilin Wu. 2018. What is decidable about string constraints with the ReplaceAll function. *PACMPL* 2, POPL (2018), 3:1–3:29.

[13] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[14] ECMA International. 2015. *ECMAScript 2015 Language Specification.*

[15] Xiang Fu, Michael C. Powell, Michael Bantegui, and Chung-Chih Li. 2013. Simple linear string constraints. *Formal Asp. Comput.* 25, 6 (2013).

[16] Patrice Godefroid. 2007. Compositional Dynamic Test Generation. In *ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL)*.

[17] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*.

[18] Patrice Godefroid, Michael Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing. In *Network and Distributed System Security Symp. (NDSS)*.

[19] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015. DLint: Dynamically Checking Bad Coding Practices in JavaScript. In *Int. Symp. on Software Testing and Analysis (ISSTA)*.

[20] Lukás Holík, Petr Janku, Anthony W. Lin, Philipp Rümmer, and Tomás Vojnar. 2018. String constraints with concatenation and transducers solved efficiently. *PACMPL* 2, POPL (2018), 4:1–4:32.

[21] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. In *ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*.

[22] Guodong Li, Esben Andreasen, and Indradeep Ghosh. 2014. SymJS: automatic symbolic testing of JavaScript web applications. In *Foundations of Software Engineering (FSE)*.

[23] Guodong Li and Indradeep Ghosh. 2013. PASS: String solving with parameterized array and interval automaton. In *Haifa Verification Conf. (HVC)*.

[24] Nuo Li, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. Reggae: Automated test generation for programs using complex regular expressions. In *Automated Software Engineering (ASE)*.

[25] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. 2014. A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions. In *Computer Aided Verification (CAV)*.

[26] Tianyi Liang, Nestan Tsiskaridze, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. 2015. A Decision Procedure for Regular Membership and Length Constraints over Unbounded Strings. In *Int. Symp. on Frontiers of Combining Systems (FroCoS)*.

[27] Blake Loring, Duncan Mitchell, and Johannes Kinder. 2017. ExpoSE: Practical Symbolic Execution of Standalone JavaScript. In *Int. SPIN Symp. on Model Checking Software (SPIN)*.

[28] Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*.

[29] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A Symbolic Execution Framework for JavaScript. In *IEEE Symp. Sec. and Privacy (S&P)*.

[30] Joseph D. Scott, Pierre Flener, and Justin Pearson. 2015. Constraint Solving on Bounded String Variables. In *Integration of AI and OR Tech. in Constraint Prog. (CPAIOR)*.

[31] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Foundations of Software Engineering (FSE)*.

[32] Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel C. Briand. 2017. Search-driven string constraint solving for vulnerability detection. In *Int. Conf. Software Engineering (ICSE)*.

[33] Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized unit tests. In *Foundations of Software Engineering (FSE)*.

[34] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In *Conf. Computer and Commun. Sec. (CCS)*.

[35] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2016. Progressive Reasoning over Recursively-Defined Strings. In *Computer Aided Verification (CAV)*.

[36] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2017. Model Counting for Recursively-Defined Strings. In *Computer Aided Verification (CAV)*.

[37] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. 2010. Rex: Symbolic regular expression explorer. In *Software Testing, Verification and Validation (ICST)*.

[38] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Murphy Berzish, Julian Dolby, and Xiangyu Zhang. 2017. Z3str2: an efficient solver for strings, regular expressions, and length constraints. *Formal Methods in System Design* 50, 2-3 (2017).

[39] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Julian Dolby, and Xiangyu Zhang. 2015. Effective Search-Space Pruning for Solvers of String Equations, Regular Expressions and Length Constraints. In *Computer Aided Verification (CAV)*.

[40] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: A Z3-based String Solver for Web Application Analysis. In *Foundations of Software Engineering (FSE)*.